SOLO: Self Organizing Live Objects

Qi Huang[1, 2] (contact, tel. +16073519956), Ken Birman[2]
[1]*School of Computer Science & Technology*
Huazhong Univ. of Science & Technology
qihuang@hust.edu.cn
[2]*Department of Computer Science*
Cornell University
{qhuang, ken}@cs.cornell.edu

Abstract

Data dissemination overlays are central in scalable multicast protocols and are used in many other kinds of distributed systems. Such overlays must self-assemble, and in situations where there are multiple protocol options, a suitable choice of protocol may be key to achieving desired levels of performance, reliability, or other QoS objectives. Here, we describe SOLO, a new platform we're constructing as part of Cornell's Live Objects project. SOLO automates the task of discovering the runtime environment by sensing such properties as NAT or firewall characteristics, bottlenecks and bandwidth fluctuations, etc. This paper presents the SOLO architecture and evaluates its effectiveness under a range of realistic scenarios that would be expected in wide-area environments.

Category: Regular paper for the PDS track (5700 words).
Keywords: Self-organization, Live Objects, Multicast, Distributed Systems, Overlay Networks
This submission is NOT subject to any institutional release or publication restrictions.

1. Introduction

By making it possible to efficiently transmit packets to potentially large numbers of destinations, multicast dissemination patterns arise in systems that stream media files, replicate data, or support collaborative work. However, because the Internet WAN has evolved to optimize support for TCP and point-to-point UDP, multicast generally operates as an end-to-end service, supported by packages that struggle to work around such barriers as firewalls, network bottlenecks and NAT boxes. There exists a large body of work on multicast scalability [17] and application layer multicast (ALM) [18], but each solution tends to have its own special environmental assumptions. Deploying multicast in real WAN settings remains difficult.

Yet there has never been a wider variety of interesting multicast data distribution schemes. When available, router-supported IP multicast can provide high reliability [7] [19] and optimization for mobile devices in LAN settings [13]. Overlay-based ALM has become common in WAN data-streaming platforms, and some applications built over these capabilities are extremely popular, for example in support of large-scale content distribution. In settings where IP multicast is permitted, protocols that exploit it can achieve exceptional performance and scalability.

These developments establish the context for our work. SOLO, a new platform on which we report here, assists application designers in delaying choices: an application can incorporate more than one possible transport protocol, selecting the appropriate communication infrastructure for their application based on runtime conditions that our tools discover automatically. SOLO also assists the application in configuring the selected communication layer to conform to performance and topology properties of the network, and can orchestrate adaption if conditions change.

SOLO runs in potentially complex environments, and while multicast is a primary focus, the system can also be used for other kinds of overlays or transport choices. WAN systems make heavy use of NAT boxes, and various firewall settings can block the basic packet transfer between hosts. Communication protocols that are blind to such barriers will perform poorly or fail. Moreover, unstable bandwidth and delay is common in edge networks, and both sometimes fluctuate over time; for some multicast algorithms, these kinds of issues can defeat optimization decisions or trigger failures. Host mobility is a growing challenge: when a host moves from one access point to another, configuration changes may be needed.

These are not insurmountable challenges. Products such as BitTotrrent [5] and PPLive [6] have established that a sufficiently sophisticated designer can take many cases into account. But unless we can find ways to reuse the needed infrastructure, such solutions benefit just a single application. Moreover, if multiple appli-

Report Documentation Page

Form Approved
OMB No. 0704-0188

1. REPORT DATE	2. REPORT TYPE	3. DATES COVERED
DEC 2008		**00-00-2008 to 00-00-2008**

4. TITLE AND SUBTITLE	5a. CONTRACT NUMBER
SOLO: Self Organizing Live Objects	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S)	5d. PROJECT NUMBER
	5e. TASK NUMBER
	5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) **Cornell University,Department of Computer Science,Ithaca,NY,14853**	8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSOR/MONITOR'S ACRONYM(S)
	11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited

13. SUPPLEMENTARY NOTES

14. ABSTRACT

Data dissemination overlays are central in scalable multicast protocols and are used in many other kinds of distributed systems. Such overlays must selfassemble and in situations where there are multiple protocol options, a suitable choice of protocol may be key to achieving desired levels of performance, reliability or other QoS objectives. Here, we describe SOLO, a new platform we?re constructing as part of Cornell?s Live Objects project. SOLO automates the task of discovering the runtime environment by sensing such properties as NAT or firewall characteristics bottlenecks and bandwidth fluctuations, etc. This paper presents the SOLO architecture and evaluates its effectiveness under a range of realistic scenarios that would be expected in wide-area environments.

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			
unclassified	**unclassified**	**unclassified**	**Same as Report (SAR)**	**10**	

cations run on the same node, there may be a great deal of redundant overhead as each struggles to detect its environment and to monitor for changes.

SOLO is not the first system to help application developers solve these kinds of problems. For example, BBN's QuO [8] extends CORBA with a quality of service architecture. Developers document a set of runtime cases and, for each, its performance needs using the QuO interface language. The runtime system automatically detects conditions, determines which case applies, and the components configure themselves appropriately; if conditions later change, QuO sends events that notify the components and can trigger reconfiguration.

However, our problem is harder than the one solved by QuO, which ran in enterprise LAN settings. It is much more difficult to talk about system-wide performance characterization in WANs, where conditions may vary drastically from region to region and where there is no easy way to collect metrics at a single server. Moreover, even for a single WAN application, performance needs can vary in use-dependent ways. For example in a content sharing system, the lag-tolerance for a live-streaming use may depend on the kind of content being streamed.

SOLO operates in stages. It allows a system to specify a set of possible multicast components before deployment. At runtime, SOLO instruments the network and selects the best multicast components within the set. It then provides continuous information updates, helping the system detect and react to such problems as a loss of bandwidth on a bottleneck link on which a group of hosts depends. By standardizing the model and associated events, we help the developer understand what conditions should be handled and the styles of dynamic adaptation to consider using.

The SOLO mechanisms operate locally, at the end-hosts where the application runs. Although it would be interesting to explore mechanisms for unifying end-host data into some form of shared distributed database, we leave this and the mechanisms it would enable for future work. For the present, communication protocols constructed incorporate their own mechanisms for sharing SOLO's node-status information.

SOLO was built using Cornell's Live Objects platform (LO) [11], [12]. LO is a component-oriented architecture that permits distributed applications to be constructed as mashups (graphs) of components, described in XML "recipes" and then shared via files or email. SOLO runs as an LO application, and leverages many LO features. Nonetheless, SOLO can be used as a standalone service by a multicast platform otherwise oblivious to the LO infrastructure.

The contributions of this paper are as follows:

- We present an environment-aware architecture for designing multicast systems.
- We provide a set of tools that automate detection and monitoring of the network, with functionality aimed at the application designer, protocol deployment and runtime adaptation.
- We show how the componentized Live Objects architecture was used to simplify SOLO.

2. SOLO Architecture

This section starts with a brief overview of the SOLO architecture, and then gives a more detailed description of its major components.

2.1. Multicast Components

SOLO makes some assumptions about the protocol components with which it operates. First, it assumes that these were co-designed and intended to operate *side by side*. For example, a multicast option set might include an ALM for use in the Internet WAN and an IP multicast-based protocol for use when peers find themselves in a LAN setting that supports IP multicast. Individual end-hosts might find that only the ALM version can be used, or might be capable of launching both. When multiple transports are used by a single application, the transport layer (not SOLO) must relay messages between protocol instances.

SOLO also assumes that each protocol has a bootstrapping mechanism, whereby protocol components can locate peers and establish initial contact with them (perhaps directly, or perhaps through a reachback site in the Internet). SOLO itself provides no membership tracking, but once components have an initial peering relationship, SOLO can help them learn about one-another's environment, as discussed in Section 3.

The assumptions just described are well matched to the behavior of multicast platforms such as PPLive [6] and AnySee [15], on which one of us (Huang) was a lead developer. Indeed, we restructured code from these to undertake the evaluation reported in Section 4.

2.2 SOLO Overview

SOLO has three components: the Local Detection Service, the Multicast Selector and the Deployment Helper. As shown in Figure 1, the Multicast Selector is a kind of rule-driven component library filter for the system designer. With it, an application is able to specify a set of transport protocols and the conditions under which each can be used.

The Deployment Helper selects the available multicast components based on detected conditions, and then provides the component with configuration help to assist it during initialization and drive adaptation if conditions change. The local detection service tests the environment. For example, it discovers bottleneck links, and if a node has interfaces on multiple networks, it determines the connectivity of each interface.

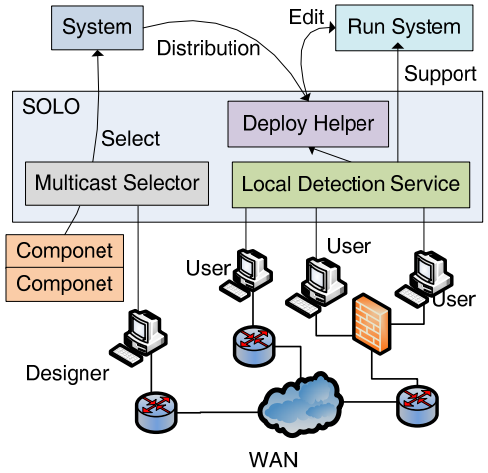

Figure 1. SOLO architecture

SOLO's Local Detection is focused on the state of the host network interfaces, NAT and firewall settings, and the performance of the deployed network. It works by pinging nodes on the path from the end host to the closest DNS. Previous studies have suggested that this approach has a high likelihood of revealing connectivity limits and bottleneck links [1]. Below, we'll see how information collected this way can also be used to group nodes co-located behind a shared bottleneck.

2.2. Multicast Selector

In SOLO, multicast components are only launched if they find themselves in a compatible runtime environment. As noted earlier, rather than cluttering a single multicast protocol with all sorts of adaptively selected configuration options, SOLO encourages the designer to build a set of protocols, each with its own relatively rigid runtime requirements, but integrated to be capable of running side-by-side.

SOLO's Multicast Selector uses a set of rules to decide which protocols to launch. Each includes a *Communication Restriction* and (optionally) a *Performance Requirement.* A Communication Restriction is represented as a quadruple (*D, T, P, O*). In each quadruple, *D* specifies the direction of communication to be tested, and *T* the transport layer protocol (currently, TCP or UDP). *P* selects a specific application protocol, and *O* is used to designate the existence of performance requirements. The available value for each field is listed in Table 1.

Table 1. Communication Restriction value

Fields	Values	Meaning
D	*O*	Outward only
	A	All by default
T	*TCP*	TCP only
	UDP	UDP only
	A	All by default
P	*S*	Permit specified only
	A	Permit all
O	*Y*	Do need performance
	N	No specific performance

The restriction associated with *P* gives the rule designer access to additional runtime tests. For instance, if the designer includes a protocol component that operates using HTTP messages for use in networks where the NAT and firewall policy permit such messages to reach the end-host, the quadruple should be (*O, TCP, S, N*), and an extra parameter to *P* would tell SOLO that the required protocol is HTTP. Similarly, SOLO can test for connectivity to a rendezvous node, or to servers in a data center. SOLO can support the full range of WAN tests that are used to configure PPLive and AnySee.

SOLO's Performance Requirement rules describe network performance constraints. These are used to rule out protocol choices that would fail for lack of bandwidth, excessive latency, or other readily detectable problems. These rules take the form of a list of tests described by three properties: (*F, RL, RH*). *F* represents the performance metric of interest, and *RL~RH* specify the acceptable range of performance values, as illustrated in Table 2. For example, the designer of an overlay-multicast voice conference system could specify a requirement like (*D, 0, 30ms*) to encode the minimal voice QoS properties from [16].

Table 2. Performance Requirement value

Fields	Values	Meaning
F	*D*	Delay
	B	Bandwidth
	L	Loss rage
RL	*$set*	Range lower limit
	N	No lower limit
RH	*$set*	Range upper limit
	N	No upper limit

SOLO's performance tests are all *local*, evaluated by measuring performance of the network route from the local host to its DNS server. Thus, even if two hosts both conclude that some protocol can be instantiated, it might still be unable to directly connect to some of the other hosts running the same application.

2.3. Deployment Helper

After filtering, each SOLO-equipped host will initialize one or more protocol components, as seen in Figure 2. In this example, a multicast application with components on hosts A and C has concluded that only one of the available multicast transports was suitable on each, but host B is running both, and will relay messages between them.

For interface standardization, when multiple protocols are available, they will typically implement a single *interface type*. For the experiments reported later in this paper, we defined a base *chunk-pull* overlay multicast interface with four messages types: *Buffer Snapshot*: exchange the buffer condition; *Peer Information*: contains the peer address to communicate; *Data Request*: requests data in a pull manner; and *Data Chunk*: which contains the distributed data. We then built several multicast protocols, each inheriting from this shared interface type.

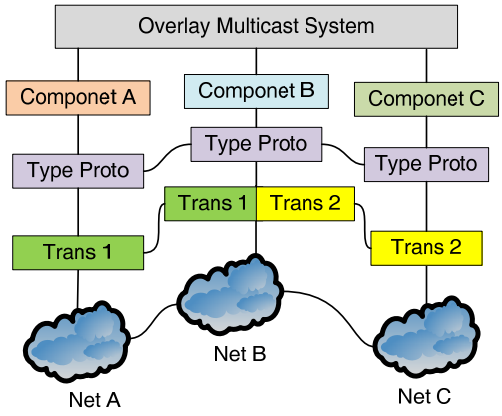

Figure 2. Component variation in network

The SOLO Deployment Helper is used to configure communication components. For example, suppose that a transport layer communication method is sensitive to NAT and firewall settings. Even though nodes A and B may both conclude that component X can be used, X may still be prevented from making direct peering connections between A and C. To assist protocols in detecting such problems, SOLO builds a list of *EUIDs* (Environment Unique Identities) that represent the communication characteristics of each host. The *EUID* data is maintained locally at end hosts using SOLO, but protocol components can share this data with peers (either during their bootstrap exchange, and then later during normal execution). Thus, when A attempts to peer with C, it will have a set of *EUIDs* describing C's local environment.

The *EUID* contains the environment information detected locally including the Communication Restriction and network interface information, as shown in Figure 3. The {Pub Addr} field represents the NAT address visible to external hosts, while the {Proto ID} points to the firewall allowable application layer protocol if {P} is set to value *S* (as Table 1).

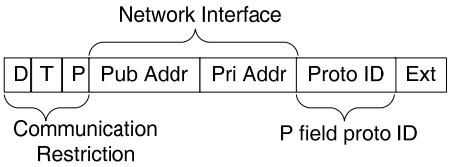

Figure 3. EUID structure

Given a set of *EUIDs* for a remote host with which it needs to communicate, a SOLO-equiped node can compare the remote *EUID* data with its own to identify communication options that might work. For example, with reference to Figure 2, host A cannot talk directly to host C, but is able to peer with host B using transport component 1, and C can peer with B using transport 2.

2.4. Local Detection Service

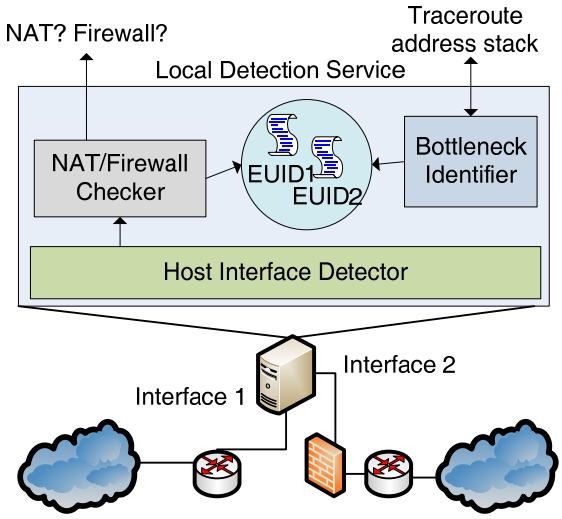

Figure 4. Local Detection Service components

The technical core of SOLO is its local detection service. The service is composed of two parts: the Host Interface Detector and the Bottleneck Identifier.

2.4.1. Host Interface Detector. The role of this module is to collect data from each network interface, tabulating it into a form from which *EUID* records can be constructed. The *EUID* is defined on a per-interface basis because, in large deployments, it is common to encounter multi-homed hosts. For example, routers and multi-ISP homed servers almost always have multiple interfaces. We see this in Figure 5, where Node A has this sort of multi-homed configuration; the *Whois* utility can be used to discover that the same machine holds different addresses in two ISPs.

Failing to distinguish between interfaces can lead to errors. For example, many multicast systems use the host name to acquire the local interface address to bind, or simply bind on all the interfaces using zeroes in the address field of the bind argument. Such approaches have hidden problems. The former method may lead failure or inefficiency of connection attempts from remote hosts which are actually nearer to some other unbound interface in the routing table. Simply binding on all interfaces may open an unsecure port between external network and private network.

By treating local interfaces separately, SOLO is able to describe a host in sufficient detail to permit correct configuration decisions. We note that multi-homing information is used in many ALM and VOIP protocols. For example, ALMs often route data through different ISPs to improve performance and availability, and VOIP protocols often do so to get the lowest possible delays.

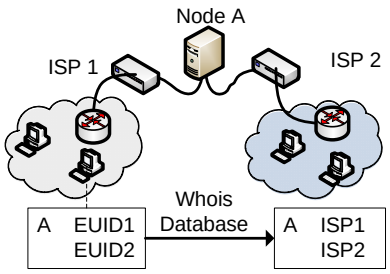

Figure 5. Bridge of multi-interface host

Another role of the Host Interface Detector is to sense access-point changes for mobile devices. When a mobile device migrates between wireless routers, the device's *EUID* values will often change. Protocol configurations that are keyed by the *EUID* value will unambiguously identify the router the protocol is trying to use. Finally, the Host Interface Detector checks for a NAT/Firewall and senses the associated settings, using the method detailed in [4].

2.4.2. Bottleneck Identifier. It is common to find groups of hosts that reside behind a bottleneck link on the edge network, especially in a home network or a small corporate LAN. Few overlay multicast systems are able to optimize themselves for such cases, because they lack the needed environment information to detect them. Thus, in current overlay multicast platforms, it is not uncommon to see heavily traffic interference on the shared link. In section 4, we demonstrate this problem experimentally. The work described in [20] also identified this phenomenon, which it models as a correlation called LLC.

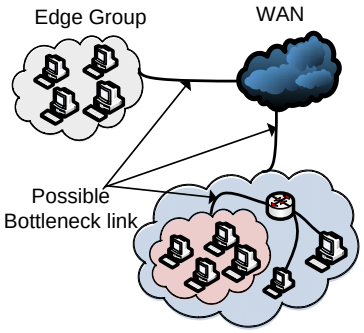

Figure 6. Clustered edge peer group

SOLO's Bottleneck Identifier helps multicast components identify possible bottleneck link by grouping peers on different sides of it. We believe our approach to be a new one. Unlike the work of [2] [9], which focuses on direct measurement of end-to-end bandwidth, SOLO clusters peers into groups as shown in Figure 6, and suggests the overlay multicast component best suited for use under the detected conditions.

The key challenge is to recognize peers in the same group. For peers behind the same NAT, it is natural and simple to group them by checking public IP address of NAT box. But this is not a complete solution since an edge network may have multiple public routable IP addresses, while sharing a single link.

Some Peer-to-Peer systems use the public IP prefix to identify clusters of peers. But this approach is too coarse-grained to find a bottleneck link because some ISPs assign the same IP prefix to machines distributed over large geographical areas [14]. Another solution is to use the local DNS server address, a method common in CDN systems. However, this method is also known to be too coarse-grained. For example, studies by AT&T [21] found that only 69% of hosts in the same

autonomous systems (ASs) are configured to use the local DNS servers.

In SOLO, we use traceroute to detect the routing addresses between gateway and the local DNS server, with max hop count (TTL) set to 4. This is surprisingly effective, and confirms an early study [10] in which it was found that network latency is dominated by the last few hops from the core Internet to a destination host. Accordingly, SOLO constructs an address stack 4 hops deep and includes this into the *EUID*. Given a pair of *EUIDs* for distinct hosts, it is now possible to group them, as illustrated by the tree structure shown in Figure 7.

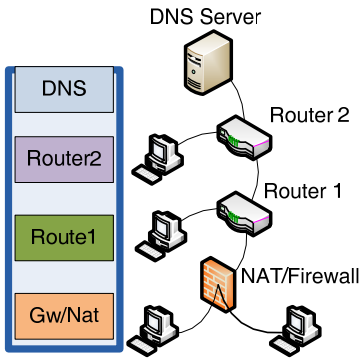

Figure 7. Address stack and grouping tree

A protocol component using SOLO can employ this address stack to detect possible bottlenecks. For example, based on the grouping tree, a multicast component can configure itself to aggregate and evaluate the bandwidth between different levels of groups. If the data transfer between two levels reveals a high loss rate, the link shared by the inner group might be a bottleneck link. Moreover, bandwidth fluctuation can be also discovered. In section 4, we show an experiment in this method groups local peers to optimize streaming performance despite the limitations imposed by a bottleneck link.

3. Implementation based on Live Objects

In this section, we provide additional details concerning our implementation of SOLO within the existing Live Objects platform. We start with some basic background concerning the Live Objects work, and then focus on SOLO per-se.

3.1. Live Distributed Objects

SOLO was constructed using the Windows version of Live Objects, and was coded in the .NET C# language. The live objects platform was created as a tool for supporting distributed and often collaborative applications that might mix content hosted on web services platforms with event streams or other media implemented through direct peer-to-peer protocols. Once created, a live object has an XML representation that can be stored in files, sent over email, etc. Users who activate the same live object will typically see instances of some form of reliable, consistent, distributed functionality.

An individual live object represents some form of reusable protocol or application component, but it is rare for a live object to be used in isolation. More often, objects are composed and interoperate with other live objects. Thus, a typical application will be a graph of connected objects: a “mashup”. Such a mashup, as it might look on a single host, is shown in Figure 8.

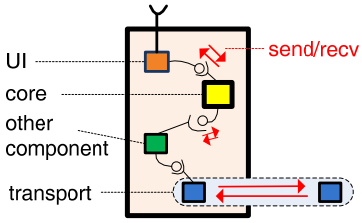

Figure 8. Structure of a live objects application

Notice that the bottom object in the application, the “transport” protocol, peers with other instances of the same object on remote machines. SOLO can be understood as selecting one or more such live distributed protocol objects, and (in effect) reconfiguring the mashup dynamically on the basis of the environment.

Communication between live objects involves passing typed events over connected endpoint pairs. We see some examples of endpoints in Figures 8 and 9. As in other component-based systems (including Microsoft’s .NET platform, on which Live Objects was initially based), each endpoint consists of a collection of function interfaces that can be invoked asynchronously. The live objects platform performs type checking, determines which function to invoke when events cross interfaces, and glues the objects together.

The key strengths of the live objects model stem from its flexibility. For example, live objects can integrate “pure edge” protocols, such as data replication, with content pulled from “cloud computing” platforms, such as web services systems that support the usual SOA standards. Additionally, live objects have a representation that can be shared through an XML encoding.

SOLO leverages the live objects type-checking mechanisms. For each communication protocol, SOLO generates endpoint references that encode the informa-

tion in the Communication Restrictions and Performance Requirement rule sets, representing these are endpoint type constraints. At runtime, SOLO generates an additional endpoint, annotated with the discovered properties of the environment. The live objects type-checking algorithm will then select matching endpoint(s), activating only the protocol components that satisfy the local restrictions.

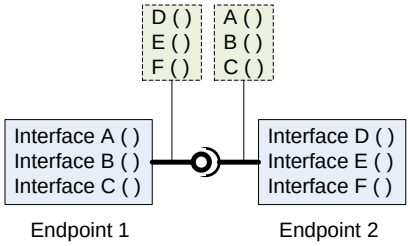

Figure 9. Two connected Endpoints

3.2. Rules for Filtering Multicast Components

To illustrate these ideas, we describe the way that we use the mechanism to select an appropriate multicast protocol. Table 3 shows the matched endpoints to Communication Restriction and Performance Requirement. In Table 3, the Fields *D*, *T* and *P* are matched with the same fields in Communication Restriction. When they are set to *A*, Multicast Selector will skip checking the endpoints. The last field is for Performance Requirement. Figure 10 illustrates the process by which rule-checking selects the appropriate modules.

Table 3. Endpoints for Communication Restriction and Performance Requirement

Fields	Values	Endpoint Interface
D	*O*	[D:O] OnDisableContacted ( )
	A	
T	*TCP*	[T:TCP] OnDisableUDP ()
	UDP	[T: UDP] OnDisableTCP ()
	A	
P	*S*	[P: S: Protocol ID] OnPrivProtocolParser ( )
	A	
F *RL* *RH*		[F: RL: RH] OnOutRange ( )

Recall from Section 2.2 that after filtering the available protocol modules, SOLO passes a list of EUID values to the multicast modules, which share these during their bootstrapping protocol. The desired effect is that peers can compare EUID values and thus agree on the best option for connecting. In some situations, this will mean that a single host ends up running multiple protocols side-by-side.

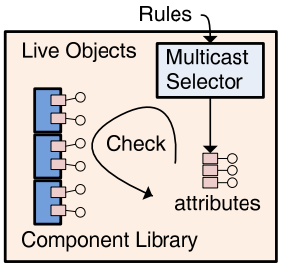

Figure 10. Process of checking rules

3.3. Multiplexer

When a host runs multiple transport protocols, SOLO assists them in handling the incoming traffic using a communication *multiplexer*. This runs beneath the multicast transports, vectoring messages to the appropriate component, and partially automates the selection of the best communication route to use.

The Multiplexer is designed to listen on three kinds of ports. Most protocols use some mixture of dedicated TCP and UDP sockets, and the multiplexer can monitor these if desired (for example, if two multicast protocol modules share a single UDP port). Additionally, WAN protocols used in systems such as AnySee often include a fall-back HTTP web-services scheme for use when all else fails. As shown in Figure 11, the multiplexer inserts a header of its own on outgoing traffic and interprets the headers of incoming messages, vectoring them appropriately. Internally, the parser used for HTTP ports is implemented as a live object, and could be replaced with other objects to support other forms of last-resort rendezvous/tunneling solution (e.g. via a database, IM, or email system).

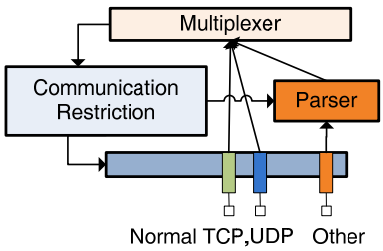

Figure 11. Multiplexer

4. Experiments on SOLO Detection Service

We designed an experiment to illustrate the effectiveness of the SOLO platform, focusing initially on the performance of the SOLO Local Detection Service. This experiments was run on the Emulab testbed [3] at Utah and focuses on our running example: a multicast transport that uses SOLO to adaptively configure itself.

4.1. Bottleneck Influence on Multicast

Recall that in section 2.4.3, we emphasized the sensitivity of current overlay based multicast to bottleneck link performance. To establish ground-truth, we began by designing an experiment to illustrate this issue, as a function of bottleneck topology. We studied a network consisting of two sets of hosts connected with a 1.5Mbps link. One set of hosts represents the External 100Mbps WAN and the other represents the local group behind the bottleneck link connected in the same 100Mbps LAN. The overlay multicast algorithm employed is a swarm-style mesh-based overlay that disseminates data using a pull-based local-rarest-first rule. Our experiment models a streaming download in which a source node generates a 1Mbps bit-rate stream, seeding randomly selected clients with chunks of data that they then collaborate to share. The data consists of a series of 180 "stream buffers" each containing 1 second of video content. The chunk size was set to 0.2 seconds of data, and each experiment lasts for 20 minutes. A stream buffer must be completely received on time; partial buffers are discarded.

We simulated a setup in which nodes reside either in the Internet WAN (W) or within several LANs (L), each situated behind a bottleneck. Lacking any form of locality awareness, download swarms will attempt to fetch chunks over the bottleneck links more or less at random. Our bottlenecks can serve no more than two streams concurrently, hence if too much remote communication occurs, download quality suffers. This is visible in Table 4, where at most 13% of buffers were successfully received with 5 LANs and 5 nodes in each.

Table 4. Isolated group experiment result

# of W	# of L	# in L	Avg. Buffer Percentage
4	*5*	5	12.695%
4	*1*	20	4.089%
16	*1*	20	3.978%

The rest two rows also show that the download speed is only affected by the number of nodes in the same LAN, representing the severity of confliction at the bottleneck link.

Figure 12 looks more closely at the average buffer quality as a function of time. We look at hosts situated in five LANs, with 4 additional hosts in the Internet WAN (row 1 of Table 4). Initially, the LAN hosts buffers fill nearly linearly, reaching 20%. However, as nodes begin to reach out for rare chunks, they stress the bottleneck links, and most buffers are still incomplete when the timeout expires.

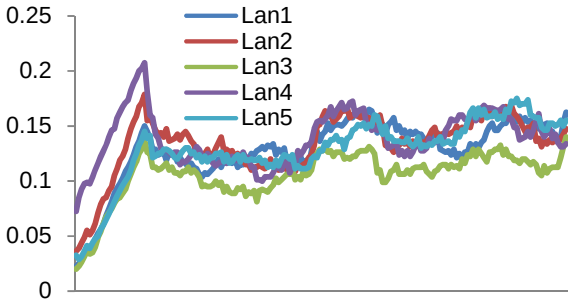

Figure 12. Buffer condition of isolated 5 LANs with 4 WAN hosts

Table 5. Completed group experiment result

# of W	# of L	# in L	Avg. Buffer Percentage
4	*5*	5	35.869%
4	*1*	20	24.827%
16	*1*	20	15.993%

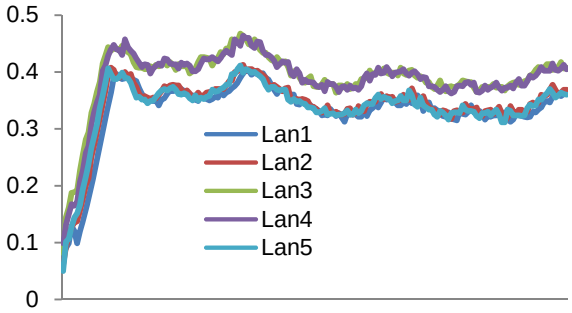

Figure 13. Buffer condition of locality biased 5 LANs with 4 WAN hosts

It is natural to wonder if the locality-aware overlay mechanisms that have been proposed for BitTorrent and PPLive would improve these results. Accordingly, we repeated our experiment with same environment settings, but using locality-aware biasing mechanisms. Table 5 and Figure 13 show the results. We can see that with bias in favor of nearby chunks the bottleneck stress problem is delayed, and the overall success rate

for buffer reception rises. Yet as illustrated in Figure 13, the bottleneck remains a problem: after each node in a LAN has obtained all the chunks accessible from other hosts in the same LAN, app nodes all turn towards WAN data sources for rare chunks, overloading the bottleneck link.

4.2. Optimization Result based on SOLO

We next modified our system to use the tools provided by SOLO. SOLO is able to distinguish between LAN and WAN nodes, and peers within the same region can discover this by comparison of the address stacks encoded into the *EUID* structure. Recall that during the bootstrapping stage, peers will have discovered one-another and shared *EUID* information. On the other hand, because SOLO lacks a replication mechanism of its own, applications that share *EUID* data must be aware that it could become stale as nodes join, leave, or move about.

Given the possibility of recognizing of co-located nodes, it is easy to build a bottleneck-link aware variant of the swarming protocol. In this scheme, nodes issue chunk-requests preferentially to nodes in the same group. A single *group leader* is the only one to request data from outside the group. Notice that this scheme should be quite robust even if *EUID* data evolves at runtime.

Table 6. Completed group experiment result

# of W	# of L	# in L	Avg. Buffer Percentage
4	*5*	5	70.453%
4	*1*	20	72.660%
16	*1*	20	66.452%

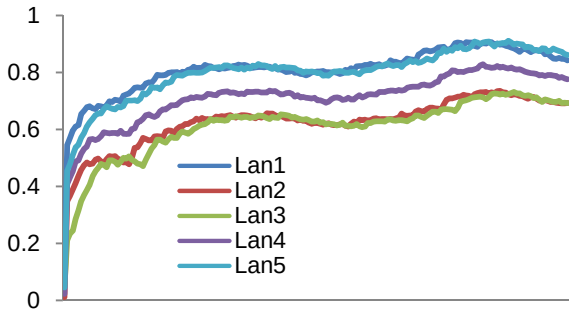

Figure 14. Buffer condition of optimized 5 LANs with 4 WAN hosts

Table 6 and Figure 14 show the results of this modified experiment, still running in the same configuration. Our network is still overloaded, and we see that not all buffers are successfully received. However, the percentage success rate has risen to between 66% and 73%. There are many streaming media applications that would be unusable with a 13% success rate, but acceptable at 66%.

Notice that since only one leader exists in each LAN group, the number of hosts that reside with it behind the same bottleneck link should not (and does not) affect the quality of data transfer across the link. Thus, to the extent that the SOLO heuristics are able to detect bottleneck links and correctly identify co-located hosts, SOLO-assisted performance should be sharply better than what one sees in more traditional BitTorrent or PPLive swarm-style downloads.

4.3. Network Interface and NAT/Firewall Checking

For reasons of brevity, we were not able to present experiments to evaluate the network interface detection and NAT/Firewall detection mechanisms supported by SOLO. Nonetheless, even in the previous experiments, SOLO's network interface detection plays a useful role. As readers may be aware, Emulab machines have six network interfaces, one on the 159.98.36 network. SOLO's interface detection mechanism was used to ensure that the overlay multicast used all active interfaces except 159.98.36 network.

5. Conclusion and Future Work

Our paper presented SOLO, a live-objects based system to assist transport protocols in configuring themselves to cope with potentially challenging network conditions. SOLO has a rich set of functionality, supporting a wide range of adaptations that are seen in systems such as BitTorrent and PPLive. Our experiments focused on one novel capability enabled by SOLO: detection of bottleneck links that partially isolate groups of nodes. We demonstrated that by exploiting such information, one can create streaming multicast protocols that are more effective than protocols that fail to do so. Down the road, we plan to explore a wider range of adaptations, and support for non-multicast applications such as VOIP.

6. Acknowledgements

We are grateful to the Chinese National Research Foundation, the National Science Foundation, the Air Force Research Laboratories at Rome NY, Intel and Cisco for their support of this research.

7. References

[1] A. Akella, S. Seshan, and A. Shaikh, "An Empirical Evaluation of Wide-Area Internet Bottlenecks", *In Proceedings of IMC'03*, Miami, Florida, USA, October, 2003.

[2] Allen B Downey, "Using pathchar to estimate Internet link characteristics", In Proceedings of SIGCOMM'99, Cambridge, MA, USA, 1999.

[3] B. White, J. Lepreau, L. Stroller, R. Ricci, S. Guruprasad, M. Newbold, M. Hibler, C. Barb, and A. Joglekar. "An intergrated experimental environment for distributed systems and networks", *In Proceedings of OSDI'02*, 2002.

[4] Bryan Ford, Pyda Srisuresh, and Dan Kegel, "Peer-to-Peer Communication Across Network Address Translators", *In Proceedings of USENIX*, Anaheim, CA ,2005

[5] http://www.bittorrent.com

[6] http://www.pplive.com

[7] J.C. Liu and S. Paul, "RMTP: a reliable multicast transport protocol. *In Proceedings of IEEE Infocom'96*, San Francisco, USA, March 1996.

[8] John A. Zinky, Davie E. Bakken and Richard E. Schantz, "Architectural Support for Quality of Service for CORBA Objects", *Theory and Practice of Object System*, Jan 1997.

[9] Kevin Lai and Mary Baker, "Nettimer: A Tool for Measuring Bottleneck Link Bandwidth", *Proceedings of the USENIX Symposium on Internet Technologies and Systems*, March 2001.

[10] Krishna P. Gummadi, Stefan Saroiu, and Steven D. Gribble, "King: Estimating Latency between Arbitrary Internet End Hosts", *In Proceedings of SIGCOMM Internet Measurement Workshop*, Marseille, France, November 2002.

[11] Krzysztof Ostrowski, Ken Birman, Danny Dolev, "Live Distributed Objects: Enabling the Active Web", *IEEE Internet Computing*, November, 2007.

[12] Krzysztof Ostrowski, Ken Birman, Danny Dolev, and Jong Hoon Ahnn, "Programming with Live Distributed Objects", *In Proceedings of ECOOP*, Cyprus, July 2008.

[13] Lin, C. R., Kai-Min Wang, "Mobile multicast support in IP networks", *In Proceedings of INFOCOM*, Tel Aviv, Israel, 2000.

[14] Michael J. Freedman, Mythili Vutukuru, Nick Feamster and Hari Balakrishnan, "Geographic Locality of IP Prefixes", *In Proceedings of IMC'05*, Berkeley, California, USA, October 2005.

[15] Qi Huang, Hai Jin, and Xiaofei Liao, "P2P Live Streaming with Tree-Mesh based Hybrid Overlay", *In Proceedings of ICPP Worshop'07*, Xi'an, China, 2007.

[16] Shansi Ren, Lei Guo, and Xiaodong Zhang, "ASAP: an AS-Aware Peer-Relay Protocol for High Quality VoIP", *In Proceedings of ICDCS'06*, Lisboa, Portugal, July 2006.

[17] Stephen E. Deering, David R. Cheriton, "Multicast routing in datagram internetworks and extended LANs", *ACM Transactions on Computer Systems*, Volume 8, Issue 2, pp. 85-110, May 1990.

[18] Suman Banerjee, Bobby Bhattacharjee, Christopher Kommareddy, "Scalable Application Layer Multicast", *In Proceedings of SIGCOMM'02*, Pittsburgh, Pennsylvania, USA, August, 2002.

[19] S. Floyd, V. Jacobson, C. Liu, S. McCanne, and L. Zhang, "A reliable multicast framework for light-weight sessions and application level framing", *IEEE/ACM Transactions on Networking*, 5(6), December 1997.

[20] Ying Zhu, Baochun Li, "Overlay Multicast with Inferred Link Capacity Correlations", *In Proceedings of ICDCS'06*, Lisbon, Portugal, July 2006

[21] Zhuoqing Morley Mao, Charles D. Cranor, Fred Douglis, and Michael Rabinovich, "A Precise and Efficient Evaluation of the Proximity between Web Clients and their Local DNS Servers", *In Proceedings of USENIX'02, Monterey, California*, June 2002.